Recovering Commit Dependencies for Selective Code Integration in Software Product Lines

Tejinder Dhaliwal, Foutse Khomh, Ying Zou
Dept. of Elec. and Comp. Engineering
Queen's University
Kingston, Ontario, Canada
{9td23, foutse.khomh, ying.zou}@queensu.ca

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
ahmed@cs.queensu.ca

***Abstract*—In software product lines, multiple products of a software product family, share source code of common components. New features added to the common components of a software product family, are integrated into products following a selective code integration process. Selective code integration is a process in which developers pick the commits (*i.e.*, code changes) related to a feature from one code branch and integrate them into another code branch. Developers often manually link the commits to the features to enable the selective integration of features. In current practice, not all dependent commits are always linked to features and developers might miss the unlinked commits during selective code integration. In this paper, we propose two grouping approaches that identify dependencies among commits and create groups of dependent commits that need to be integrated as a whole into a code branch. Our first approach is automatic and the other is developer-guided. Through a case study on data derived from a product line of mobile software applications, we show that our approaches can achieve a precision of up to 95% and a recall of up to 82% in grouping dependent commits. We also show that our approaches can help to reduce by up to 94% integration failures caused by missing commit dependencies.**

***Keywords*-Selective Code Integration; Developer-guided Grouping; Product Line Development; Empirical Software Engineering.**

I. Introduction

Version Control Systems (VCS) allow development teams to track and manage source code changes. Modern VCS use a change oriented model [1]. In such a model, every code change submitted by a developer is stored as a *commit*. New commits are added to the source code to implement Change Requests (CR). A CR is a call for a modification of a system. The modification can be, for example, an addition of a feature or a bug fix.

VCS support *branching*, which allow separate changes to be applied to different copies of a project. Each copy is called a *code branch* and changes from one branch can be integrated into another branch. When all the commits from one branch are integrated into another branch, the integration is called a *code merge*. Whereas, when a specific CR from one branch is integrated into another branch, by selectively integrating the commits related to the CR, the integration is called a *selective code integration*.

Software product lines allow developers to build similar software products (*e.g.*, multiple variants of a software), using common software artifacts [2]. A group of similar products sharing common software artifacts is known as a *product family*. In software product lines, a product family is supported by a common main branch (*i.e.*, trunk). The trunk contains the source code common to all the products in the family. For each product, a separate code branch is diverged from the trunk to store and track code changes, specific to the product. Selective code integration is a very important step in the development and maintenance of software product lines. During the development of a product family, after a new CR is added to the trunk, developers selectively integrate the CR from the trunk to the code specific to the products. During the maintenance of a product family, when a common bug is fixed on a product specific branch, developers selectively integrate commits related the bug fix into other product specific branches.

Selective code integration is a brittle process. A CR can be implemented by multiple commits sharing dependencies between them and/or depending on previous commits in the branch. Developers often manually link commits to CR to enable the selective integration of the CR. When a group of commits share dependencies among them, all the commits should be integrated as a whole into the destination branch. For example, if the implementation of a change request CR_1 modifies a common utility function on which another change request CR_2 is dependent, commits of CR_1 also need to be integrated during the integration of CR_2. However, developers often overlook or miss some commit dependencies because they are not always explicit, especially when they occur at run time (*e.g.*, dependencies between commits of CR_1 and CR_2).

When commit dependencies are missed during a selective code integration, the integration fails. This failure is characterized by either a compilation or a runtime error. We have observed from an analysis of the product line of a mobile software application that integration failures caused by missing commit dependencies can increase the total integration time of a CR by 530%. Therefore, it is critical for development and maintenance teams to avoid missing commit dependencies during the selective integration of a CR.

In this paper, we propose two approaches to identify com-

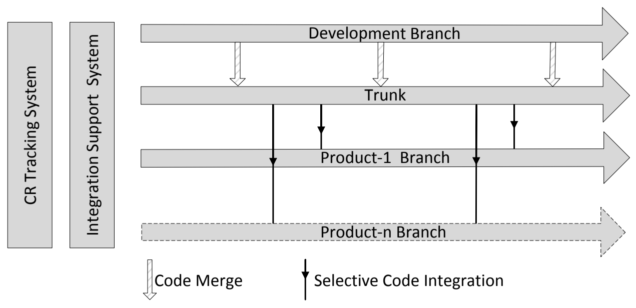

Figure 1. Version Control Model (with Selective Code Integration)

mit dependencies and create groups of dependent commits that should be integrated together. The first approach is an automatic grouping approach that assists developer to find the group of commits that needs to be integrated when they are integrating a CR. The second approach is a developer-guided approach that assists developers in finding a group of related commits to a particular commit. These approaches are particularly important because in large software applications, developers generally have limited knowledge of the complete application [3] and usually the developer responsible for the integration of a CR is not the one who implemented the CR. Therefore, it is important to assist developers in determining commit dependencies. Our two approaches determine dependencies among the commits by analyzing dependencies among CRs, structural and logical dependencies among source code elements, and the history of developers' working collaborations.

This paper makes the following three contributions:

- We propose four dissimilarity metrics to recover dependency links between commits.
- We propose two approaches to group the dependent commits, in order to assist developers during selective code integration process.
- We evaluate our proposed approaches on data from a product line of mobile software applications and show that they can help to avoid 94% of the integration failures caused by missing commit dependencies.

The rest of the paper is organized as follows. Section II gives an overview of the selective code integration process. Section III describes our approaches for grouping dependent commits. Section IV presents the setup of our case study to validate our proposed grouping approaches. Section V discusses the results of the case study. Section VI discusses the threats to the validity of our study. Section VII summarizes the related literature on the selective code integration process in Software Product Lines. Finally, Section VIII concludes our work.

II. Overview of a Selective Code Integration Process

The selective integration of CRs in software product lines requires the following three essential systems:

1) **The CR Tracking System** (*e.g.*, Jira[1]) which tracks the CRs of each version of the products. This system also helps developers to maintain the dependencies among the CRs. A change request CR_1 is considered dependent on another change request CR_2, if for example, CR_1 makes use of a feature introduces by CR_2.
2) **The Version Control System** (*e.g.*, Git[2]) which maintains source code branches. Software product lines often follows a main line branching model [4], as shown in Figure 1. This model contains three types of source code branches: the development branch, the trunk and the product specific branches. The development branch and the trunk are common to all the products in a product family, with each product having its own branch.
3) **The Integration Support System** (*e.g.*, Gerrit[3]) which is used to facilitate the selective code integration. The integration support system maintains the links between the commits and the CRs. There are two types of links:
   - ***Commit to CR links*** - When developers implement a CR, they link related commits with the CR.
   - ***Commit to commit links*** - When a commit added for one CR, is dependent on another commit added for a different CR, developers link the two commits.

   The Integration support system also keeps track of the integration state of each CR. A CR can have the following integration states:
   - ***Ready for Integration*** - When a CR is implemented and merged into the trunk, the CR is marked as *ready for integration* into product branches.
   - ***Integrated*** - When a CR is successfully integrated into a product branch, the CR is marked as *integrated*.
   - ***Failed to Integrate*** - When the integration of a CR into a product branch fails, the CR is marked as *failed to integrate*.

A typical selective code integration process involves two types of actors:

1) **Developer** - A developer commits the code changes for a CR into the development branch and integrates the CR into the trunk. Developers implement features for all the products.
2) **Integrator** - An integrator belongs to a specific product team. Integrators selectively integrate the CRs from the trunk into their product specific branches.

Figure 2 shows the sequence of a selective code integration process. The following actions are performed during the selective integration of a CR:

- The developer implements the CR, commits the code changes into the development branch and links the commits with the CR. A CR can be implemented by one or multiple developers.
- When a CR is implemented and tested in the development branch, it is merged into the trunk and marked as ready

1. www.atlassian.com/software/jira
2. www.git-scm.com/
3. www.code.google.com/p/gerrit/

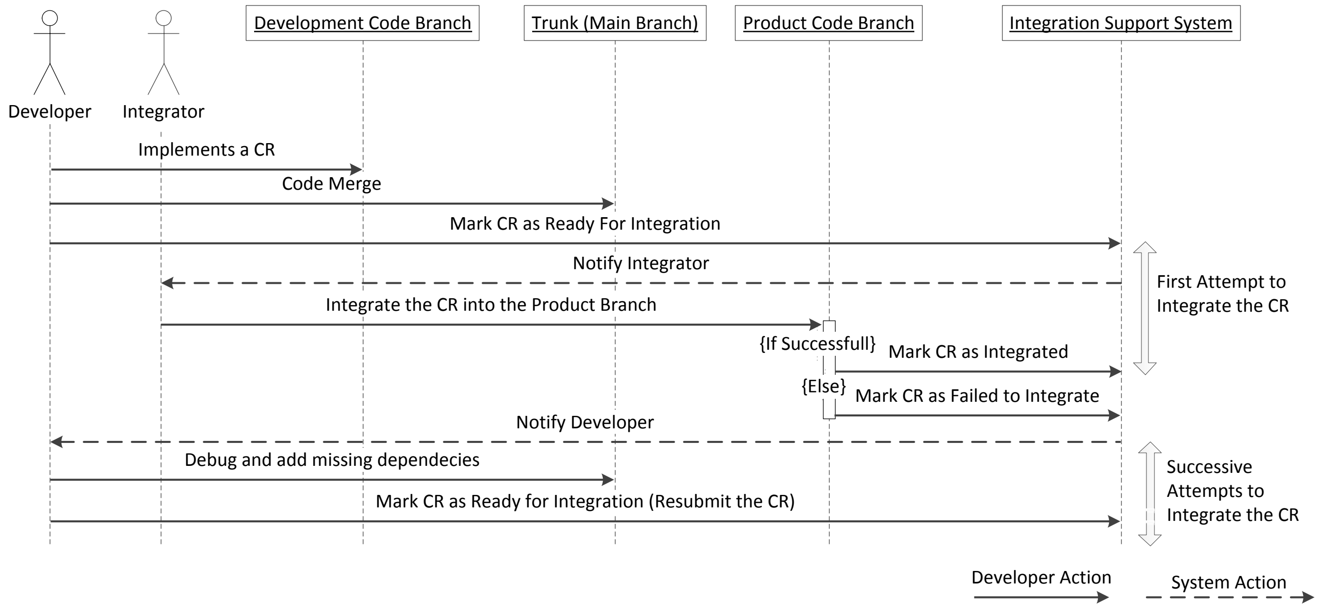
Developer
Integrator
Development Code Branch
Trunk (Main Branch)
Product Code Branch
Integration Support System
Implements a CR
Code Merge
Mark CR as Ready For Integration
Notify Integrator
First Attempt to Integrate the CR
Integrate the CR into the Product Branch
{If Successfull}
Mark CR as Integrated
{Else}
Mark CR as Failed to Integrate
Notify Developer
Debug and add missing dependecies
Successive Attempts to Integrate the CR
Mark CR as Ready for Integration (Resubmit the CR)
Developer Action
System Action

Figure 2. Sequence Diagram of the Selective Integration of a Change Request.

for integration (into product specific branches).

- Once a CR is marked as ready for integration, all product integrators are notified. Each integrator checks if the CR is applicable for their product. If applicable, the integrator integrates all the commits linked to the CR into the product specific branch.
- When integrating a CR, all commits linked directly or indirectly with the CR are integrated into the product branch. For example, if some commits of a change request CR_1 are linked with commits of a change request CR_2, and the change request CR_1 is dependent on another change request CR_3 (in the CR tracking system), the integration of CR_1 requires that all commits linked with CR_1, CR_2 and CR_3 are to be integrated all together.
- When the integration succeeds, the CR is marked as integrated and the process is completed. Otherwise, the CR is marked with an integration failure notice and the developers who implemented the CR are notified.
- Developers should debug the source code related to the CR, make the required changes and/or add the missing commit dependencies, and resubmit the CR for integration.

III. Grouping Of Commits

When integrating a CR into a branch, developers have to incorporate all the commits linked with the CR. However, all too often some commits linked with a CR depend on other commits not directly linked with the CR. Sometimes, commit dependencies are dynamic *i.e.*, a source code of two commits are dependent only at runtime. Developers often have a hard time tracking indirect or dynamic dependencies when integrating CRs. Missing dependencies cause integration failures. To assist developers during CRs integration and prevent failures caused by missing dependencies, we propose two approaches to identify dependencies between commits. The approaches also group dependent commits together and link them to the corresponding CRs.

The remainder of this section discusses the details of our identification of commits dependencies and introduces the two grouping approaches.

A. Identification of Commit Dependencies

A commit is a set of source code changes submitted by a developer into a code branch. A commit contains information such as the list of files affected by the changes, the name of the developer submitting the changes, the date of submission, the description of the changes, and other information about the branch in which the changes are submitted. In this work, we represent a commit using a set of three attributes: the list of files modified in the commit, the name of the developer who submitted the commit, and the CR linked with the commit. We select these attributes because they contain essential information about the code elements that have been changed and the authors of the changes.

To identify dependencies between commits, we introduce the following four dissimilarity metrics.

1) ***File Dependency Distance*** *(FD):* The file dependency distance (FD) measures the dissimilarity between two commits using source code dependencies between the classes contained in the files modified in the commits.

If $C_a = \{F_a, D_a, CR_a\}$ and $C_b = \{F_b, D_b, CR_b\}$ represents two commits; with $F_a =< f_{a1}, \dots f_{an} >$ and $F_b =< f_{b1}, \dots f_{bm} >$ being respectively the lists of modified files in C_a and C_b; D_a and D_b the names of the developers who submitted C_a and C_b respectively; and CR_a and CR_b the CRs linked with C_a and C_b respectively, we define FD in Equation

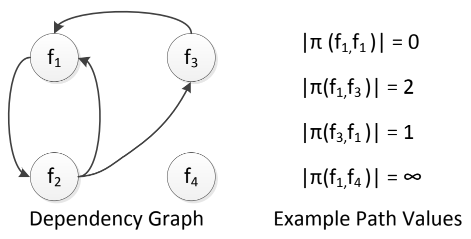

Figure 3. Source Dependency Graph

(1).

$$FD(C_a, C_b) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} (1 - Dp(f_{ai}, f_{bj}))}{n * m} \quad (1)$$

Where $Dp(f_{ai}, f_{bj})$ measures the level of structural dependency between f_{ai} and f_{bj}.

A java file f_a is considered dependent on a java file f_b, if a class defined in f_a instantiates, invokes a function from, extends, or implements a class defined in f_b. To extract dependencies between files in an application, we parse the source code of the application and build a file dependency graph. Figure 3 shows an example of the file dependency graph. Each node in a file dependency graph represents a source code file. A directed link between two nodes means that one file is dependent on the other file. We use dependency graphs to calculate the level of structural dependency between two files, f_a and f_b, following Equation (2).

$$Dp(f_a, f_b) = \frac{1}{2}[\frac{1}{1 + |\pi(f_a, f_b)|} + \frac{1}{1 + |\pi(f_b, f_a)|}] \quad (2)$$

Where $|\pi(f_a, f_b)|$ is the number of nodes on the shortest path from f_a to f_b in the dependency graph.

We consider both $|\pi(f_a, f_b)|$ and $|\pi(f_b, f_a)|$ in the definition of $Dp(f_a, f_b)$ because dependency graphs are directed graphs.

*2) **File Association Distance** (FA):* In the definition of FD, the dependency between two files is measured using structural dependencies between the java classes defined in the files. However, two files can be dependent without a direct source code dependency, for example two files that changed together are considered logically dependent [5]. We introduce the File Association (FA) distance to capture logical dependencies between commits. We use the mean square contingency coefficient (ϕ) [6] to measure the level of logical association between files in commits. The ϕ coefficient is a measure of association between two events, *e.g.*, the modification of two files f_a and f_b. The value of ϕ is computed using Equation (3).

$$\phi(f_a, f_b) = \frac{(p_1 p_4 - p_2 p_3)}{\sqrt{(p_1+p_2)*(p_3+p_4)*(p_2+p_4)*(p_1+p_3)}} \quad (3)$$

Where, p_1 is the number of commits in which both f_a and f_b were modified, p_2 is the number of commits in which f_a was modified but not f_b, p_3 is the number of commits in which f_b was modified but not f_a, and p_4 is the number of commits in which neither f_a nor f_b were modified.

$\phi(f_a, f_b)$ captures the co-occurrence of modifications of f_a and f_b. A value $\phi = 0$ indicates that there is no logical association between f_a and f_b, while a value $\phi = 1$ indicates a perfect association (*i.e.*, f_a and f_b are always modified together).

The FA distance of two commits $C_a = \{F_a, D_a, CR_a\}$ and $C_b = \{F_b, D_b, CR_b\}$, where $F_a =< f_{a1}, \dots f_{an} >$ and $F_b =< f_{b1}, \dots f_{bm} >$, is defined in Equation (4).

$$FA(C_a, C_b) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} (1 - \phi(f_{ai}, f_{bj}))}{n * m} \quad (4)$$

*3) **Developer Dissimilarity Distance** (DD):* Often, dependent commits are submitted by developers who were collaborating on CRs in the past. We introduce the Developer Dissimilarity Distance (DD) to capture the working relation between two developers submitting commits. Similarly to FA, we use the ϕ coefficient to measure the working relation between two developers. We define the ϕ coefficient for two developers D_a and D_b following Equation (5).

$$\phi(D_a, D_b) = \frac{(p_1 p_4 - p_2 p_3)}{\sqrt{(p_1+p_2)*(p_3+p_4)*(p_2+p_4)*(p_1+p_3)}} \quad (5)$$

Where p_1 is the number of CRs implemented by both D_a and D_b, p_2 is the number of CRs in which D_a contributed but not D_b, p_3 is the number of CRs in which D_b contributed but not D_a, and p_4 is the number of CRs in which neither D_a nor D_b contributed.

The DD distance between two commits $C_a = \{F_a, D_a, CR_a\}$ and $C_b = \{F_b, D_b, CR_b\}$, submitted respectively by developers D_a and D_b is defined in Equation (6).

$$DS(C_a, C_b) = 1 - \phi(D_a, D_b) \quad (6)$$

*4) **Change Request Dependency Distance** (CRD):* The Change Request Dependency Distance (CRD) captures the dissimilarity between the CRs implemented by two commits. When two commits are linked with CRs marked (by developers) as dependent, the CRD value is 0. Otherwise the CRD value is 1. Equation (7) gives a formalization of the CRD between two commits, C_i and C_j.

$$CRD(C_i, C_j) = \begin{cases} 0 & \text{if } C_i \text{ and } C_j \text{ are linked with} \\ & \text{the same CR or dependent CRs} \\ 1 & \text{otherwise} \end{cases} \quad (7)$$

B. Commit Grouping Approaches

Figure 4 gives an overview of our process to identify, group, and link dependent commits to the corresponding CR. The process is composed of three parts. The first part consists in learning levels of dissimilarity among dependent commits using historical commit data and metrics from Section III-A. The second part describes a process to assign a commit to existing groups of dependent commits. The third part defines the grouping algorithms for each of our two approaches. The algorithms are based on the dissimilarity levels learned in the first part. Both algorithms also use the commit assignment process from the second part. In the following we discuss each part in details.

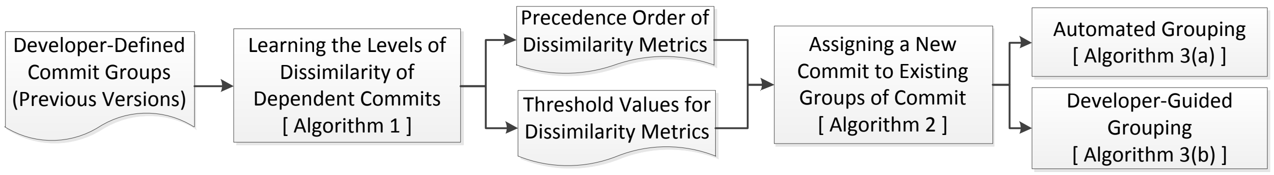
Developer-Defined Commit Groups (Previous Versions)
Learning the Levels of Dissimilarity of Dependent Commits [Algorithm 1]
Precedence Order of Dissimilarity Metrics
Threshold Values for Dissimilarity Metrics
Assigning a New Commit to Existing Groups of Commit [Algorithm 2]
Automated Grouping [Algorithm 3(a)]
Developer-Guided Grouping [Algorithm 3(b)]

Figure 4. Overview of Grouping Approach

Algorithm 1: *LearnHistoryData($C[\,]$)*

```
/* Learn Dissimilarity Levels from Historical Data */
input : Commits of Previous Version $C[\,]$
output: Dissimilarity metrics $D[\,]$, in order of precedence
output: $Max\_Threshold$ and $Min\_Threshold$ values for each Dissimilarity metric
1  Organize the commits into Developer-defined commit groups ;
2  $D[\,]$ = $[FD, FA, DD, CBD]$ ;
3  foreach Dissimilarity metric $d$ in $D[\,]$ do
4      foreach Commit $C_i$ in $C[\,]$ do
5          $a_i$ = dissimilarity of $C_i$ with its own group;
6          $b_i$ = $min$ (dissimilarity of $C_i$ with other groups) ;
7          $S_d(C_i)$ = $(b_i - a_i)$ / max$(b_i, a_i)$ ;
8      $Max\_Threshold_d$ = mean value of $b_i$ $\forall C_i : C[\,]$;
9      $Min\_Threshold_d$ = mean value of $a_i$ $\forall C_i : C[\,]$;
10     $S_d$ = mean value of $S_d(C_i)$ $\forall C_i : C[\,]$;
11 Sort dissimilarity metrics in $D[\,]$ in descending order of their $S_d$ value ;
```

***Part 1: Learning the Levels of Dissimilarity of Dependent Commits**:* We learn the levels of dissimilarity among dependent and independent commits using commit data from previous versions. Algorithm 1 presents the steps of our learning process.

- Using the links (added manually by developers) between commits and CRs of previous versions, *i.e.*, commit to CR links and the CR dependency links, we organize the commits into groups. We call these groups *developer-defined groups*. [Line 1]
- $D[\,]$ is the list of dissimilarity metrics defined is Section III-A. [Line 2]
- Using *developer-defined groups* of previous versions, we compute the $Max\_Threshold$ value, the $Min\_Threshold$ value and the overall *Silhouette* Value of each dissimilarity metric. [Line 3-10]
- We order dissimilarity metrics by their Silhouette values. [Line 11]

To obtain *developer-defined groups*, we assign commits linked together (by developers) in the same group, and commits with no link to separate groups.

In summary, in our *developer-defined groups*, each group represents a set of dependent commits, that were integrated together in order to integrate the CR(s) linked to the commits in the group. During the integration process of commits contained in *developer-defined groups*, the links between the commits have been verified and corrected if necessary by the integrators. Therefore, the *developer-defined groups* are correct and complete. In this study we use *developer-defined groups* as our gold standard to learn dissimilarity levels (on prior versions of the application) and compute the performance of our grouping approaches (on future versions of the application).

To compute the *Silhouette* value [7] of a dissimilarity metric d, on *developer-defined groups*, we proceed as follows: given a commit C_i, we calculate the Silhouette $S_d(C_i)$ following as:

$$S_d(C_i) = \frac{(b_i - a_i)}{max(b_i, a_i)} \tag{8}$$

Where a_i is the average dissimilarity (i.e., d) between C_i and other commits from the same group; b_i is the minimum of average dissimilarity values between C_i and all the commits in other groups.

The silhouette value of a dissimilarity metric d is the mean of the silhouette values $S_d(C_i)$ of each commits C_i in $C[\,]$.

Silhouette value compares the distance between items in same group and the distance between the items in different groups. The range of silhouette values is from -1 to 1. A silhouette value of 0 represents a random grouping, while a silhouette value of 1 represents a perfect grouping.

***Part 2: Assigning a Commit to an Existing Groups of Commit**:* To assign a commit C into existing groups of commits $G[\,]$, we follow the steps described in Algorithm 2.

- For the commit C, we select the groups $G_{sel}[\,]$, such that for each dissimilarity metric d_j, the dissimilarity between a selected group G_i and the commit C, is lower than the $Max\_Threshold$ value of the dissimilarity metric d_j. [Line 2]
- Among the selected groups in $G_{sel}[\,]$, we find the group G_i that has the lowest dissimilarity with C, for the dissimilarity metric d_j with the highest ranking. If the dissimilarity between C and G_i is lower than the $Min\_Threshold$ value for the dissimilarity metric d_j, we assign the commit C to the group G_i and the algorithm terminates. Otherwise, we select d_j as the next dissimilarity metric in the order of precedence and repeat the comparison. [Line 3-7]

Algorithm 2: *AssignGroup(C_i,$G[\,]$)*

```
   /* Assign a Commit to Existing Groups
   of Commit */
   input: Commit $C$
   input: List of existing groups $G[\,]$
1  $D[\,]$ = Dissimilarity metrics in order of precedence;
2  $G_{sel}[\,] = \{G_i \in G[\,] \,|\, (d_j(C, G_i) <= Max\_Threshold_{dj}), \forall\, d_j : D[\,]\ \}$;
3  foreach Dissimilarity metric $d_j$ in $D[\,]$ do
4      $G_i$ = the group in $G_{sel}[\,]$ with minimum $d_j(C, G_i)$;
5      if $d_j(C,G_i) <= Min\_Threshold_{dj}$ then
6          assign $C$ to $G_i$ ;
7          exit ;
8  assign $C$ to a new group $G_n$;
9  add $G_n$ to $G[\,]$;
```

- If the commit is not assigned to any group, a new group is created for the commit. [Line 8-9]

Part 3: Grouping Dependent Commits*:* Using Algorithms 1 and 2, we propose two commit grouping approaches which can be applied incrementally on newly created commits during the development or the maintenance of a software application. The two approaches identify dependencies between commits and form groups of dependent commits that should be integrated together. Algorithm 3:(a) and Algorithm 3:(b) presents the details of the two approaches.

Algorithm 3: (a) *Automated Grouping*

```
    input: Number of Iterations $l$
    input: Commits to Group $C[\,]$
1   CommitGroups $G[\,]$ ;
2   foreach Commit $C_i$ in $C[\,]$ do
3       AssignGroup ($C_i$,$G[\,]$) ;
4   $iteration$=0 ;
5   while $iteration \leq l$ do
6       foreach Commit $C_i$ in $C[\,]$ do
7           AssignGroup ($C_i$,$G[\,]$) ;
8       if no commit moved from one group to another
        then
9           break;
10      $iteration$++;
```

(a) *Automated Grouping*: When an integrator integrates a CR, he or she searches for all the commits related to the CR that are available in the trunk. The trunk contains commits linked with many CRs and the links between the commits are not always explicit. In this scenario, the automated grouping approach can help integrators to recover missing dependencies among commits. The approach is presented in Algorithm 3 (a). The approach is based on the K-mean grouping algorithm [8]. The K-mean algorithm moves the elements among the groups iteratively, until a stopping criterion in met. The K-mean algorithm is not guaranteed to converge, therefore in practise a bound on the number of iterations is applied to guarantee termination [8]. For our automated grouping approach, we fixed a maximum of 1,000 iterations in case the algorithm does not terminate normally. However, we have observed through experiments that the grouping results of Algorithm 3 (a) remain similar when the maximum number of iterations is chosen between 100 and 1,000.

- For each commit C_i in $C[\,]$, we assign the commit C_i to a group in $G[]$ using Algorithm 2. Initially $G[]$ is empty. When a new commit is assigned, either a new group is created for the commit or if a group of similar commits was already created, the commit is assigned to the group of similar commits. [Line 1-3]
- Once all the commits are assigned to a group in $G[\,]$, we reassign the commits between the groups in $G[\,]$ by using Algorithm 2. During the reassignment of the commits in $G[]$, the groups are changed and a commit can move from one group to another. If no commit is moved from one group to another group during a reassignment, the algorithm terminates, otherwise we reassign the commits again. To guarantee a convergence, the algorithm is terminated after a maximum number of iterations (*i.e.*, 1,000 iterations).

(b) *Developer-Guided Grouping*: When a developer adds a new commit, he or she links the commit with a CR. However, a commit for a CR can be dependent on some other commit(s) from another CR. In this scenario, we recommend the developer-guided approach. The approach is presented in Algorithm 3 (b).

Algorithm 3: (b) *Developer-Guided Grouping*

```
   input: Newly added commit $C_n$
   input: Existing Group of Commits $G[\,]$
1  AssignGroup ($C_n$,$G[\,]$);
2  A developer verifies the grouping of $C_n$. If incorrect,
   the developer manually assigns $C_n$ to the correct
   group;
```

- When a new commit is added, the commit is assigned to a group using Algorithm 2. [Line 1]
- After the new commit is assigned to a group, a developer verifies if the commit is assigned correctly. If not, the developer manually assigns the commit to the correct group. [Line 2]

IV. Case Study Design

To show the benefits of using our grouping approach, we perform a case study using a large scale software product family. The goal of this case study is two-fold: (1) validate

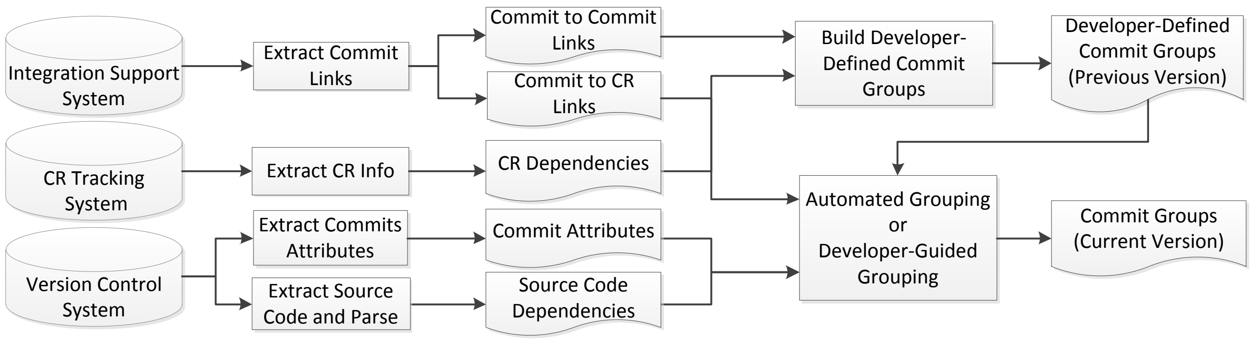
Integration Support System
Extract Commit Links
Commit to Commit Links
Commit to CR Links
Build Developer-Defined Commit Groups
Developer-Defined Commit Groups (Previous Version)
CR Tracking System
Extract CR Info
CR Dependencies
Automated Grouping or Developer-Guided Grouping
Commit Groups (Current Version)
Version Control System
Extract Commits Attributes
Commit Attributes
Extract Source Code and Parse
Source Code Dependencies

Figure 5. Data Collection and Commit Grouping

our dissimilarity metrics; (2) evaluate the effectiveness of our proposed grouping algorithms. We formulate the following research questions:

- *RQ1: Can the FD, FA, DD and CRD metrics, be used to group dependent commits?*
- *RQ2: How efficient are our proposed grouping approaches?*
- *RQ3: What is the impact of using our approach on a selective integration process?*

A. Data Collection

In this study we analyze three major versions of a family of mobile applications. The product family follows a main line branching development model (illustrated by Figure 1). We study the three most recent consecutive versions of the product family. We name these versions V1, V2 and V3.

B. Data Processing

Figure 5 gives an overview of our data processing. First, we extract CRs from the CR Tracking Systems. Then, we download the source code of V1, V2 and V3 from the Version Control System and extract dependencies between files. For each CR of either version V1, V2 and V3, we retrieve the states (*i.e.*, *Ready for Integration*, *Integrated*, *failed to integrate*) of the CR and the list of commits linked to the CR in the Integration Support System. Using this data we apply our grouping algorithms to automatically group dependent CRs and assess the effectiveness of our grouping approach. The remainder of this section elaborates on each of these data processing steps.

1) Mining CR Tracking System: We retrieved the CRs of versions V1, V2 and V3, and the dependencies among the CRs for each version using the data collection API provided by the CR Tracking System.

2) Mining the Version Control System: On the source code of each downloaded version, we extract source code dependencies between every pair of files. Using the data collection API provided by the Version Control System, we retrieve commit logs. We parse these commit logs to extract the following attributes for each commit: the list of file modified in the commit and the name of the developer who submitted the commit.

3) Mining the Integration Support System: The Internal Integration system used by the enterprise was developed internally. The system is integrated with the Version Control System to allow developers to link commits to related CRs. Using the API provided by the Integration Support System, we retrieved all the commits to CR links, the commit to commit links, and the history of the states of all the CRs from versions V1, V2 and V3.

V. Case Study Results

This section presents and discusses the results of our three research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion of our findings.

RQ1: Can the FD, FA, DD and CRD metrics, be used to group dependent commits?

Motivation: In Section III-A we proposed four metrics to measure similarities between commits. The effectiveness of the grouping approach presented in this paper is dependent on the ability of the proposed metrics to identify dependent commits. In this research question, we evaluate the effectiveness of each of our four metrics in identifying dependent commits.

Approach: We evaluate each of our metrics using the *developer-defined groups* of versions V1, V2 and V3, defined in Section III-B. For each metric and for each pair of commit from the *developer-defined groups*, we compute a dissimilarity value using the metric. We expect that two commits from the same group will have a low dissimilarity value, while commits from separate groups a high dissimilarity value. To measure the goodness of fit of our metrics to the *developer-defined groups*, *i.e.*, the ability of our metrics to identify two commits that are linked together, we compute silhouette values for each metric following Equation (8) from Section III-B. A positive silhouette value means that the metric can successfully identifies two linked commits. The higher the silhouette value, the stronger is the ability of the metric to identify linked commits.

Findings: **The CRD metric is the strongest at indicating linked commits.** Table I shows the Silhouette values of the four dissimilarity metrics measured on commits from our three studied versions. CRD has the highest Silhouette and the values are close to 1. This result was expected since one

Table I
SILHOUETTE VALUES OF THE PROPOSED DISSIMILARITY METRICS

Dissimilarity Measure	V1	V2	V3
CR Distance (CRD)	0.94	0.96	0.96
File Association Distance (FA)	0.76	0.79	0.67
Developer Dissimilarity Distance (DD)	0.63	0.67	0.57
File Dependency Distance (FD)	0.46	0.47	0.49

expect commits linked together to belong to the same CR or to dependent CRs. The FA metric has the next highest values, indicating that files from dependent commits are frequently changed together. The next in the precedence order is DD, indicating that dependent commits are often added by developers with a history of joint work on CRs. The FD metric is reported to be the least effective of our four metrics, with an average silhouette value of 0.47. However, the average FD dissimilarity between two dependent commits is 0.24 and the average FD dissimilarity between two commits with no link is 0.47. Indicating that files modified in dependent commits have slightly stronger source code dependencies compare to files modified in independent commits (*i.e.*, commits with no link). Nevertheless, FD remains the least effective of our four dissimilarity measures.

RQ2: How efficient are our proposed grouping approaches?

Motivation: In Section III-B, we have defined two approaches to group dependent commits, in order to assist a developer during a selective integration process. However, because the prime motivation for automating the grouping of commits is to save developers' time and resources, the efficiency of the proposed approaches is very important. In this research question we evaluate how correctly and how efficiently the proposed approaches can group dependent commits.

Approach: To evaluate the efficiency of our proposed automatic and developer-guided approaches, we first use *developer-defined groups* from version V1 to learn dissimilarity levels necessary to calibrate the grouping algorithms. We then apply the two grouping algorithms on the commits of V2. Second, we calibrate the two grouping algorithms using *developer-defined groups* from version V1 and V2, and we apply the grouping algorithms on the commits of version V3.

To assess the correctness of the groups created by our two algorithms, we use *developer-defined groups* of versions V2 and V3 as our gold standard. We compare how well the commit dependency links recovered by our grouping approaches (*i.e.*, the groups created by Algorithm 3(a) or Algorithm 3(b)) match the links established in the Integration Support System by the developers of V2 and V3 (*i.e.*, the *developer-defined groups*).

We compute the precision and the recall of the automated grouping approach (*i.e.*, Algorithm 3) using respectively Equation (9) and Equation (10). The precision value measures the fraction of retrieved commit dependency links that are correct, while the recall value measures the fraction of correct commit dependency links that are retrieved.

Table II
PRECISION, RECALL AND ACCURACY OF THE GROUPING APPROACHES

	V2	V3
Precision (Automated)	92%	95%
Recall (Automated)	79%	82%
Accuracy (Developer-Guided)	96%	98%

$$Precision = \frac{|\{correct\ link\} \bigcap \{retrieved\ link\}|}{|\{retrieved\ link\}|} \quad (9)$$

$$Recall = \frac{|\{correct\ link\} \bigcap \{retrieved\ link\}|}{|\{correct\ link\}|} \quad (10)$$

In the case of our developer-guided approach (*i.e.*, Algorithm 3), we use *developer-defined groups* to simulate the feedbacks from developers. Specifically, we proceed as follow: after adding a commit, we compare the grouping result of our algorithm with the groups in *developer-defined groups*. If the commit is not assigned to the correct group of dependent commits, we override the result of our algorithm and assign the commit to the correct group suggested by *developer-defined groups*. Because our developer-guided approach modifies its grouping results to match with the *developer-defined groups*, the precision and recall values at the end of the developer-guided grouping process are 100%.

To assess the effectiveness of Algorithm 3 in assigning commits to their appropriate groups, we use the accuracy metric defined in Equation (11). The accuracy measures the fraction of commits that are assigned correctly (*i.e.*, that didn't need to be corrected using developers' feedback).

$$Accuracy = \frac{|\{commits\ assigned\ correctly\}|}{|\{all\ commits\}|} \quad (11)$$

Findings: Table II shows the precision and recall values for the automated grouping approach and the accuracy values for the developer-guided approach for versions V2 and V3. When applying the grouping approaches to the commits of version V2 (respectively version V3), we used historical information from version V1 (respectively versions V1 and V2). The results for version V3 are better than the results for version V2, indicating that a longer history can help improve the effectiveness of Algorithm 3 and Algorithm 3.

1) Complexity of the grouping approaches: Our two proposed approaches use Algorithm 2 to assign a commit to a group. This algorithm compares a given commit with all the existing groups. The maximum number of commits groups can be same as the number of commits, therefore the complexity of the algorithm is $O(n)$, where n is the number of commits to be grouped. In developer-guided approach when a developer adds a new commit, the Algorithm 2 is executed to find the group of dependent commits, therefore the complexity of the developer-guided approach remains $O(n)$.

In the automated grouping approach, Algorithm 2 is applied for each existing commit. The complexity for running Algorithm 2 for each commit is $O(n^2)$. If any commit moves from one group to another, the process is repeated.

Therefore the overall complexity of the automated approach is $O(l.n^2)$, where l is the number of iterations. We have limited the number of iterations to 1,000 iterations, which restricts the complexity of the automated approach to $O(n^2)$. Moreover, the automated grouping approach is based on the K-mean algorithm and the number of iterations of the K-mean algorithm depends on the selection of the initial element of each group [9]. In our automated grouping approach, the initial groups are created by Algorithm 2, which ensures that only the independent commits are selected as initial element of a group.

RQ3: What is the impact of using our approach on a selective integration process?

Motivation: On average, after a CR is added to the trunk, it takes 6 days to integrate the CR to a production branch. But, when the integration of a CR fails, it takes 32 days to reintegrate the CR to the same production branch. Integration failures are often caused by missing commit dependencies or incorrect implementations of the CRs. When a CR fails to be integrated, the CR is returned to a developer. The developer debugs/corrects the implementation of the CR and re-submits the CR for integration. On average, an integration failure increases the integration time of a CR by 28 days. We want to evaluate if our proposed grouping approaches can reduce integration failures caused by missing dependencies between commits.

Approach: To resolve an integration failure caused by missing dependencies, developers link some existing commits with the failing CR. Sometimes, a new commit is created and linked to the CR. The CR is then resubmitted for integration.

We assess the effectiveness of our grouping approaches in preventing integration failures by computing a Failure Reduction Rate (FRR). The FRR measures the fraction of missing dependencies between commits (responsible of integration failures) that are successfully recovered by our grouping approaches. To compute FFR, we identify all CRs that ever failed during an integration (*i.e.*, CRs that ever had the state *Failed to Integrate*); second, we select among the obtained CRs, those for which the failure was caused by missing dependencies; third we compute for each selected CR (cr), the set M_{cr} of missing dependencies responsible for the failure. For each of our grouping approaches, we compute FRR using Equation (12).

$$FRR = \frac{|\{cr \in F \mid M_{cr} \subset R\}|}{|F|} \tag{12}$$

Where F is the set of CRs that failed during an integration because of a missing dependency; and R is the set of commit dependencies retrieved by our proposed grouping approach.

Findings: In our data set for versions V_2 and V_3, 1.7% of the CRs failed during an integration process, because of missing commit dependencies. This proportion represents many hundreds of CRs that failed to integrate in the two versions. The, FRR of our automated grouping (respectively developer-guided) approach is 76% (respectively 94%). Using our two grouping approaches, integration failures caused by missing dependencies can be avoided in up to 94% of the cases.

VI. Threats to Validity

We now discuss the threats to validity of our study following the guidelines for case study research [10].

Construct validity threats concern the relation between theory and observation. In this work, the construct validity threats are mainly due to measurement errors. We collected the data from the CR tracking system, the Version Control System and the integration support system using the API's provided by each system.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. Although we study multiple versions of a large software application, some of the findings might still be specific to the development and maintenance process of the studied software application.

Conclusion validity threats concern the relation between the treatment and the outcome. In evaluation of the developer-guided approach, we assume that if a commit is assigned incorrectly, developer will manually assign the commit to a group of dependent commits. However, if the developer doesn't reassign an incorrectly-assigned commit, the subsequent commits can be assigned to other groups than the groups which we observed in this experiment.

Threats to external validity concern the possibility to generalize our results. We have conducted the study using three versions of a large enterprise mobile software application. Gaining access to such industrial data is hard. Nevertheless, further validation on a larger set of software applications is desirable, considering applications from different domains, as well as several applications from the same domain.

VII. Related Work

A. Software Product Lines and Selective Integration:

Modern software product lines can be divided in two categories: integration-oriented product lines and open-compositional product lines [11]. The open-compositional category uses third party components to build software products. These third party components are insulated from product specific features. The integration-oriented category involves an extensive product specific development in the common components of the product family and developers need to selectively integrate features from the common components into the software products [11]. A differencing and merging technique [12], is proposed by Chen *et al.* to integrate features from one branch into another branch of a product family. The technique is based on the ADL design [13] of the product line and aims at assisting application architects, whereas our approaches are based on the source code and aim at assisting developers during code integration .

B. Recovering Commit Dependencies:

Many techniques [14], [15], [16] have been proposed to slice large software applications into modules using an analysis of source code dependencies and the history of source code fragments which are modified together. These techniques divide a software application into high-level modules, whereas our approaches identify related changes at a lower level of granularity. The works that are most closely related to our work are those of Hassan *et al.* [17], Zimmermann *et al.* [18] and Ying *et al.* [19], who propose impact analysis techniques to predict potential source code changes caused by a given change, using version control system. The techniques determine source code fragments that can potentially change because of a change made to the system. In contrast, we recover dependencies among the changes committed to the system.

VIII. Conclusion

Selective code integration is an integral part of product line development. However, selective code integration remains a risky process in which developers can miss commit dependencies. Missing commit dependencies cause integration failures which significantly delay the delivery of CRs. We propose four dissimilarity metrics that can be applied to retrieve dependencies between commits. Based on the dissimilarity levels learned from prior versions of a software application, we propose two approaches to group the dependent commits: a developer-guided approach, and an automatic approach. The developer-guided approach can be applied when a developer is adding a new commit, while the automated approach can be applied when an integrator is integrating a CR. Evaluations of our approaches on data derived from a software product line shows that the developer-guided grouping approach can reduce integration failures by 94% and that the automated grouping approach can reduce integration failure by 76%.

IX. Acknowledgement

We are grateful to Research In Motion (RIM) for providing access to the data used in the case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's software or hardware products.

References

[1] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson, "Change oriented versioning in a software engineering database," in *Proceedings of the 2nd International Workshop on Software configuration management*, ser. SCM '89. New York, NY, USA: ACM, 1989, pp. 56–65.

[2] R. van Ommering, "Building product populations with software components," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 255–265.

[3] M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" *J. Syst. Softw.*, vol. 43, no. 1, pp. 19–27, Oct. 1998.

[4] L. Wingred, "How software evolves," in *Practical Perforce, Channeling the Flow of Change in Software Development Collaboration*, J. Gennick, Ed. O'Reilly Media, 2005, pp. 176–197.

[5] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, oct. 2008, pp. 42 –46.

[6] J. Guilford, "The phi coefficient and chi square as indices of item validity," *Psychometrika*, vol. 6, pp. 11–19, 1941, 10.1007/BF02288569.

[7] P. J. and Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, no. 0, pp. 53 – 65, 1987.

[8] H. S. Christopher D. Manning, Prabhakar Raghavan, "Flat clustering," in *Introduction to Information Retrieval*. Cambridge University Press, 2008, pp. 321–343.

[9] P. S. Bradley and U. M. Fayyad, "Refining initial points for k-means clustering," in *Proceedings of the Fifteenth International Conference on Machine Learning*, ser. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 91–99.

[10] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[11] J. van Gurp, C. Prehofer, and J. Bosch, "Comparing practices for reuse in integration-oriented software product lines and large open source software projects," *Softw. Pract. Exper.*, vol. 40, pp. 285–312, April 2010.

[12] P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek, "Differencing and merging within an evolving product line architecture," in *Software Product-Family Engineering*, ser. Lecture Notes in Computer Science, F. van der Linden, Ed. Springer Berlin / Heidelberg, 2004, vol. 3014, pp. 269–281.

[13] E. Dashofy, A. van der Hoek, and R. Taylor, "A highly-extensible, xml-based architecture description language," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, 2001, pp. 103 –112.

[14] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, jun 1998, pp. 45 –52.

[15] A. Mockus and D. Weiss, "Globalization by chunking: a quantitative approach," *Software, IEEE*, vol. 18, no. 2, pp. 30 –37, mar/apr 2001.

[16] M. Kamkar, "An overview and comparative classification of program slicing techniques," *Journal of Systems and Software*, vol. 31, no. 3, pp. 197 – 214, 1995.

[17] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 284–293.

[18] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429 – 445, june 2005.

[19] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *Software Engineering, IEEE Transactions on*, vol. 30, no. 9, pp. 574 – 586, sept. 2004.